

FV316935260

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Mechanism for Providing Data Driven Command Line
Output**

Inventor(s):
Jeffrey P. Snover
Kenneth M. Hansen
Marco Chierotti

ATTORNEY'S DOCKET NO. MS1-1742US

1 **TECHNICAL FIELD**

2 Subject matter disclosed herein relates to command line environments, and
3 in particular to the output of commands entered via a command line environment.
4

5 **BACKGROUND OF THE INVENTION**

6 Many operating systems provide a mechanism for "stitching" (i.e.,
7 pipelining) multiple applications (e.g., utilities) together to create a custom, ad hoc
8 command that can be entered on a command line of the operating system.
9 Typically, the commands are used in system administration tools, such as for
10 managing system properties. Each of the "pipelined" utilities in the command
11 communicate with each other by transferring text. Thus, each utility in the
12 pipeline is responsible for parsing text that is received and for formatting text that
13 is output.

14 The formatting of the text that is output is performed by code within the
15 command and is, typically, based on interpreting switches provided on the
16 command line for the command. Thus, each command is responsible for
17 formatting and displaying the output, as desired.

18 Therefore, there is a need for a mechanism that provides enhanced
19 formatting options and does not require extensive code within the command in
20 order to provide the enhanced formatting options.
21

22 **SUMMARY OF THE INVENTION**

23 The present mechanism provides a data driven command line output within
24 an environment that supports a pipeline of object-based commands. Each object-
25 based command inputs a parseable object for processing and outputs another

1 parseable object for subsequent command processing. The mechanism is
2 operative to direct formatting and subsequent processing of the commands based
3 on a type of the incoming parseable object. Format information is obtained for the
4 type, such as shape, properties to display, and the like. The format information
5 may be specified within an XML-based document. The mechanism utilizes one or
6 more output processing commands, such as format commands, markup
7 commands, convert commands, transform commands, and out commands. These
8 output processing commands may be arranged within the pipeline in various ways
9 to achieve the desired output results.

10 **BRIEF DESCRIPTION OF THE DRAWINGS**

11 FIGURE 1 illustrates an exemplary computing device that may use an
12 exemplary administrative tool environment.

13 FIGURE 2 is a block diagram generally illustrating an overview of an
14 exemplary administrative tool framework for the present administrative tool
15 environment.

16 FIGURE 3 is a block diagram illustrating components within the host-
17 specific components of the administrative tool framework shown in FIGURE 2.

18 FIGURE 4 is a block diagram illustrating components within the core
19 engine component of the administrative tool framework shown in FIGURE 2.

20 FIGURE 5 is one exemplary data structure for specifying a cmdlet suitable
21 for use within the administrative tool framework shown in FIGURE 2.

22 FIGURE 6 is an exemplary data structure for specifying a command base
23 type from which a cmdlet shown in FIGURE 5 is derived.

24 FIGURE 7 is another exemplary data structure for specifying a cmdlet
25 suitable for use within the administrative tool framework shown in FIGURE 2.

1 FIGURE 8 is a logical flow diagram illustrating an exemplary process for
2 host processing that is performed within the administrative tool framework shown
3 in FIGURE 2.

4 FIGURE 9 is a logical flow diagram illustrating an exemplary process for
5 handling input that is performed within the administrative tool framework shown
6 in FIGURE 2.

7 FIGURE 10 is a logical flow diagram illustrating a process for processing
8 scripts suitable for use within the process for handling input shown in FIGURE 9.

9 FIGURE 11 is a logical flow diagram illustrating a script pre-processing
10 process suitable for use within the script processing process shown in FIGURE 10.

11 FIGURE 12 is a logical flow diagram illustrating a process for applying
12 constraints suitable for use within the script processing process shown in FIGURE
13 10.

14 FIGURE 13 is a functional flow diagram illustrating the processing of a
15 command string in the administrative tool framework shown in FIGURE 2.

16 FIGURE 14 is a logical flow diagram illustrating a process for processing
17 commands strings suitable for use within the process for handling input shown in
18 FIGURE 9.

19 FIGURE 15 is a logical flow diagram illustrating an exemplary process for
20 creating an instance of a cmdlet suitable for use within the processing of command
21 strings shown in FIGURE 14.

22 FIGURE 16 is a logical flow diagram illustrating an exemplary process for
23 populating properties of a cmdlet suitable for use within the processing of
24 commands shown in FIGURE 14.

1 FIGURE 17 is a logical flow diagram illustrating an exemplary process for
2 executing the cmdlet suitable for use within the processing of commands shown in
3 FIGURE 14.

4 FIGURE 18 is a functional block diagram of an exemplary extended type
5 manager suitable for use within the administrative tool framework shown in
6 FIGURE 2.

7 FIGURE 19 graphically depicts exemplary sequences for output processing
8 cmdlets within a pipeline.

9 FIGURE 20 illustrates exemplary processing performed by one of the
10 output processing cmdlets shown in FIGURE 19.

11 FIGURE 21 graphically depicts an exemplary structure for display
12 information accessed during the processing of FIGURE 20.

13 FIGURE 22 is a table listing an exemplary syntax for exemplary output
14 processing cmdlets.

15 FIGURE 23 illustrates results rendered by the out/console cmdlet using
16 various pipeline sequences of the output processing cmdlets.

17 **DETAILED DESCRIPTION**

18 Briefly stated, the present mechanism provides a data driven command line
19 output. A plurality of output processing cmdlets may be pipelined in various
20 sequences to provide a desired output result. The output processing cmdlets
21 include format cmdlets, markup cmdlets, convert cmdlets, transform cmdlets, and
22 out cmdlets. Display information is populated with formatting options for a type
23 of object. The mechanism is operative to direct formatting and subsequent cmdlet
24 processing based on the type of incoming structured data (e.g., object).

1 The following description sets forth a specific exemplary administrative
2 tool environment in which the mechanism operates. Other exemplary
3 environments may include features of this specific embodiment and/or other
4 features, which aim to facilitate outputting of formatted command line data.

5 The following detailed description is divided into several sections. A first
6 section describes an illustrative computing environment in which the
7 administrative tool environment may operate. A second section describes an
8 exemplary framework for the administrative tool environment. Subsequent
9 sections describe individual components of the exemplary framework and the
10 operation of these components. For example, the section on "Exemplary Process
11 for Displaying Command Line Data", in conjunction with FIGURES 19-23,
12 describes an exemplary mechanism for providing data driven command line
13 output.

14 **Exemplary Computing Environment**

15 FIGURE 1 illustrates an exemplary computing device that may be used in
16 an exemplary administrative tool environment. In a very basic configuration,
17 computing device **100** typically includes at least one processing unit **102** and
18 system memory **104**. Depending on the exact configuration and type of
19 computing device, system memory **104** may be volatile (such as RAM), non-
20 volatile (such as ROM, flash memory, etc.) or some combination of the two.
21 System memory **104** typically includes an operating system **105**, one or more
22 program modules **106**, and may include program data **107**. The operating system
23 **106** include a component-based framework **120** that supports components
24 (including properties and events), objects, inheritance, polymorphism, reflection,
25 and provides an object-oriented component-based application programming

1 interface (API), such as that of the .NET™ Framework manufactured by
2 Microsoft Corporation, Redmond, WA. The operating system 105 also includes
3 an administrative tool framework 200 that interacts with the component-based
4 framework 120 to support development of administrative tools (not shown). This
5 basic configuration is illustrated in FIGURE 1 by those components within dashed
6 line 108.

7 Computing device 100 may have additional features or functionality. For
8 example, computing device 100 may also include additional data storage devices
9 (removable and/or non-removable) such as, for example, magnetic disks, optical
10 disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable
11 storage 109 and non-removable storage 110. Computer storage media may
12 include volatile and nonvolatile, removable and non-removable media
13 implemented in any method or technology for storage of information, such as
14 computer readable instructions, data structures, program modules, or other data.
15 System memory 104, removable storage 109 and non-removable storage 110 are
16 all examples of computer storage media. Computer storage media includes, but is
17 not limited to, RAM, ROM, EEPROM, flash memory or other memory
18 technology, CD-ROM, digital versatile disks (DVD) or other optical storage,
19 magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage
20 devices, or any other medium which can be used to store the desired information
21 and which can be accessed by computing device 100. Any such computer storage
22 media may be part of device 100. Computing device 100 may also have input
23 device(s) 112 such as keyboard, mouse, pen, voice input device, touch input
24 device, etc. Output device(s) 114 such as a display, speakers, printer, etc. may
25

1 also be included. These devices are well know in the art and need not be
2 discussed at length here.

3 Computing device **100** may also contain communication connections **116**
4 that allow the device to communicate with other computing devices **118**, such as
5 over a network. Communication connections **116** are one example of
6 communication media. Communication media may typically be embodied by
7 computer readable instructions, data structures, program modules, or other data in
8 a modulated data signal, such as a carrier wave or other transport mechanism, and
9 includes any information delivery media. The term “modulated data signal”
10 means a signal that has one or more of its characteristics set or changed in such a
11 manner as to encode information in the signal. By way of example, and not
12 limitation, communication media includes wired media such as a wired network or
13 direct-wired connection, and wireless media such as acoustic, RF, infrared and
14 other wireless media. The term computer readable media as used herein includes
15 both storage media and communication media.

16 Exemplary Administrative Tool Framework

17 FIGURE 2 is a block diagram generally illustrating an overview of an
18 exemplary administrative tool framework **200**. Administrative tool framework
19 **200** includes one or more host components **202**, host-specific components **204**,
20 host-independent components **206**, and handler components **208**. The host-
21 independent components **206** may communicate with each of the other
22 components (i.e., the host components **202**, the host-specific components **204**, and
23 the handler components **208**). Each of these components are briefly described
24 below and described in further detail, as needed, in subsequent sections.
25

Host components

The host components 202 include one or more host programs (e.g., host programs 210-214) that expose automation features for an associated application to users or to other programs. Each host program 210-214 may expose these automation features in its own particular style, such as via a command line, a graphical user interface (GUI), a voice recognition interface, application programming interface (API), a scripting language, a web service, and the like. However, each of the host programs 210-214 expose the one or more automation features through a mechanism provided by the administrative tool framework.

In this example, the mechanism uses cmdlets to surface the administrative tool capabilities to a user of the associated host program 210-214. In addition, the mechanism uses a set of interfaces made available by the host to embed the administrative tool environment within the application associated with the corresponding host program 210-214. Throughout the following discussion, the term “cmdlet” is used to refer to commands that are used within the exemplary administrative tool environment described with reference to FIGURES 2-23.

Cmdlets correspond to commands in traditional administrative environments. However, cmdlets are quite different than these traditional commands. For example, cmdlets are typically smaller in size than their counterpart commands because the cmdlets can utilize common functions provided by the administrative tool framework, such as parsing, data validation, error reporting, and the like. Because such common functions can be implemented once and tested once, the use of cmdlets throughout the administrative tool framework allows the incremental development and test costs associated with

1 application-specific functions to be quite low compared to traditional
2 environments.

3 In addition, in contrast to traditional environments, cmdlets do not need to
4 be stand-alone executable programs. Rather, cmdlets may run in the same
5 processes within the administrative tool framework. This allows cmdlets to
6 exchange “live” objects between each other. This ability to exchange “live”
7 objects allows the cmdlets to directly invoke methods on these objects. The
8 details for creating and using cmdlets are described in further detail below.

9 In overview, each host program **210-214** manages the interactions between
10 the user and the other components within the administrative tool framework.
11 These interactions may include prompts for parameters, reports of errors, and the
12 like. Typically, each host program **210-213** may provide its own set of specific
13 host cmdlets (e.g., host cmdlets **218**). For example, if the host program is an email
14 program, the host program may provide host cmdlets that interact with mailboxes
15 and messages. Even though FIGURE 2 illustrates host programs **210-214**, one
16 skilled in the art will appreciate that host components **202** may include other host
17 programs associated with existing or newly created applications. These other host
18 programs will also embed the functionality provided by the administrative tool
19 environment within their associated application. The processing provided by a
20 host program is described in detail below in conjunction with FIGURE 8.

21 In the examples illustrated in FIGURE 2, a host program may be a
22 management console (i.e., host program **210**) that provides a simple, consistent,
23 administration user interface for users to create, save, and open administrative
24 tools that manage the hardware, software, and network components of the
25

1 computing device. To accomplish these functions, host program **210** provides a
2 set of services for building management GUIs on top of the administrative tool
3 framework. The GUI interactions may also be exposed as user-visible scripts that
4 help teach the users the scripting capabilities provided by the administrative tool
5 environment.

6 In another example, the host program may be a command line interactive
7 shell (i.e., host program **212**). The command line interactive shell may allow shell
8 metadata **216** to be input on the command line to affect processing of the
9 command line.

10 In still another example, the host program may be a web service (i.e., host
11 program **214**) that uses industry standard specifications for distributed computing
12 and interoperability across platforms, programming languages, and applications.

13 In addition to these examples, third parties may add their own host
14 components by creating “third party” or “provider” interfaces and provider
15 cmdlets that are used with their host program or other host programs. The
16 provider interface exposes an application or infrastructure so that the application
17 or infrastructure can be manipulated by the administrative tool framework. The
18 provider cmdlets provide automation for navigation, diagnostics, configuration,
19 lifecycle, operations, and the like. The provider cmdlets exhibit polymorphic
20 cmdlet behavior on a completely heterogeneous set of data stores. The
21 administrative tool environment operates on the provider cmdlets with the same
22 priority as other cmdlet classes. The provider cmdlet is created using the same
23 mechanisms as the other cmdlets. The provider cmdlets expose specific
24 functionality of an application or an infrastructure to the administrative tool
25

1 framework. Thus, through the use of cmdlets, product developers need only create
2 one host component that will then allow their product to operate with many
3 administrative tools. For example, with the exemplary administrative tool
4 environment, system level graphical user interface help menus may be integrated
5 and ported to existing applications.

6 **Host-specific components**

7 The host-specific components **204** include a collection of services that
8 computing systems (e.g., computing device **100** in FIGURE 1) use to isolate the
9 administrative tool framework from the specifics of the platform on which the
10 framework is running. Thus, there is a set of host-specific components for each
11 type of platform. The host-specific components allow the users to use the same
12 administrative tools on different operating systems.

13 Turning briefly to FIGURE 3, the host-specific components **204** may
14 include an intellisense/metadata access component **302**, a help cmdlet component
15 **304**, a configuration/registration component **306**, a cmdlet setup component **308**,
16 and an output interface component **309**. Components **302-308** communicate with a
17 database store manager **312** associated with a database store **314**. The parser **220**
18 and script engine **222** communicate with the intellisense/metadata access
19 component **302**. The core engine **224** communicates with the help cmdlet
20 component **304**, the configuration/registration component **306**, the cmdlet setup
21 component **308**, and the output interface component **309**. The output interface
22 component **309** includes interfaces provided by the host to out cmdlets. These out
23 cmdlets can then call the host's output object to perform the rendering. Host-
24 specific components **204** may also include a logging/auditing component **310**,
25

1 which the core engine **224** uses to communicate with host specific (i.e., platform
2 specific) services that provide logging and auditing capabilities.

3 In one exemplary administrative tool framework, the intellisense/metadata
4 access component **302** provides auto-completion of commands, parameters, and
5 parameter values. The help cmdlet component **304** provides a customized help
6 system based on a host user interface.

7 **Handler components**

8 Referring back to FIGURE 2, the handler components **208** includes legacy
9 utilities **230**, management cmdlets **232**, non-management cmdlets **234**, remoting
10 cmdlets **236**, and a web service interface **238**. The management cmdlets **232** (also
11 referred to as platform cmdlets) include cmdlets that query or manipulate the
12 configuration information associated with the computing device. Because
13 management cmdlets **232** manipulate system type information, they are dependant
14 upon a particular platform. However, each platform typically has management
15 cmdlets **232** that provide similar actions as management cmdlets **232** on other
16 platforms. For example, each platform supports management cmdlets **232** that get
17 and set system administrative attributes (e.g., get/process, set/IPAddress). The
18 host-independent components **206** communicate with the management cmdlets via
19 cmdlet objects generated within the host-independent components **206**.
20 Exemplary data structures for cmdlets objects will be described in detail below in
21 conjunction with FIGURES 5-7.

22 The non-management cmdlets **234** (sometimes referred to as base cmdlets)
23 include cmdlets that group, sort, filter, and perform other processing on objects
24 provided by the management cmdlets **232**. The non-management cmdlets **234**
25

1 may also include cmdlets for formatting and outputting data associated with the
2 pipelined objects. An exemplary mechanism for providing a data driven command
3 line output is described below in conjunction with FIGURES 19-23. The non-
4 management cmdlets **234** may be the same on each platform and provide a set of
5 utilities that interact with host-independent components **206** via cmdlet objects.
6 The interactions between the non-management cmdlets **234** and the host-
7 independent components **206** allow reflection on objects and allow processing on
8 the reflected objects independent of their (object) type. Thus, these utilities allow
9 developers to write non-management cmdlets once and then apply these non-
10 management cmdlets across all classes of objects supported on a computing
11 system. In the past, developers had to first comprehend the format of the data that
12 was to be processed and then write the application to process only that data. As a
13 consequence, traditional applications could only process data of a very limited
14 scope. One exemplary mechanism for processing objects independent of their
15 object type is described below in conjunction with FIGURE 18.

16 The legacy utilities **230** include existing executables, such as win32
17 executables that run under cmd.exe. Each legacy utility **230** communicates with
18 the administrative tool framework using text streams (i.e., stdin and stdout), which
19 are a type of object within the object framework. Because the legacy utilities **230**
20 utilize text streams, reflection-based operations provided by the administrative tool
21 framework are not available. The legacy utilities **230** execute in a different
22 process than the administrative tool framework. Although not shown, other
23 cmdlets may also operate out of process.
24
25

1 The remoting cmdlets **236**, in combination with the web service interface
2 **238**, provide remoting mechanisms to access interactive and programmatic
3 administrative tool environments on other computing devices over a
4 communication media, such as internet or intranet (e.g., internet/intranet **240**
5 shown in FIGURE 2). In one exemplary administrative tool framework, the
6 remoting mechanisms support federated services that depend on infrastructure that
7 spans multiple independent control domains. The remoting mechanism allows
8 scripts to execute on remote computing devices. The scripts may be run on a
9 single or on multiple remote systems. The results of the scripts may be processed
10 as each individual script completes or the results may be aggregated and processed
11 en-masse after all the scripts on the various computing devices have completed.

12 For example, web service **214** shown as one of the host components **202**
13 may be a remote agent. The remote agent handles the submission of remote
14 command requests to the parser and administrative tool framework on the target
15 system. The remoting cmdlets serve as the remote client to provide access to the
16 remote agent. The remote agent and the remoting cmdlets communicate via a
17 parsed stream. This parsed stream may be protected at the protocol layer, or
18 additional cmdlets may be used to encrypt and then decrypt the parsed stream.

19 **Host-independent components**

20 The host-independent components **206** include a parser **220**, a script engine
21 **222** and a core engine **224**. The host-independent components **206** provide
22 mechanisms and services to group multiple cmdlets, coordinate the operation of
23 the cmdlets, and coordinate the interaction of other resources, sessions, and jobs
24 with the cmdlets.
25

Exemplary Parser

The parser 220 provides mechanisms for receiving input requests from various host programs and mapping the input requests to uniform cmdlet objects that are used throughout the administrative tool framework, such as within the core engine 224. In addition, the parser 220 may perform data processing based on the input received. One exemplary method for performing data processing based on the input is described below in conjunction with FIGURE 12. The parser 220 of the present administrative tool framework provides the capability to easily expose different languages or syntax to users for the same capabilities. For example, because the parser 220 is responsible for interpreting the input requests, a change to the code within the parser 220 that affects the expected input syntax will essentially affect each user of the administrative tool framework. Therefore, system administrators may provide different parsers on different computing devices that support different syntax. However, each user operating with the same parser will experience a consistent syntax for each cmdlet. In contrast, in traditional environments, each command implemented its own syntax. Thus, with thousands of commands, each environment supported several different syntax, usually many of which were inconsistent with each other.

Exemplary Script Engine

The script engine 222 provides mechanisms and services to tie multiple cmdlets together using a script. A script is an aggregation of command lines that share session state under strict rules of inheritance. The multiple command lines within the script may be executed either synchronously or asynchronously, based on the syntax provided in the input request. The script engine 222 has the ability

1 to process control structures, such as loops and conditional clauses and to process
2 variables within the script. The script engine also manages session state and gives
3 cmdlets access to session data based on a policy (not shown).

4 **Exemplary Core Engine**

5 The core engine **224** is responsible for processing cmdlets identified by the
6 parser **220**. Turning briefly to FIGURE 4, an exemplary core engine **224** within
7 the administrative tool framework **200** is illustrated. The exemplary core engine
8 **224** includes a pipeline processor **402**, a loader **404**, a metadata processor **406**, an
9 error & event handler **408**, a session manager **410**, and an extended type manager
10 **412**.

11 Exemplary Metadata Processor

12 The metadata processor **406** is configured to access and store metadata
13 within a metadata store, such as database store **314** shown in FIGURE 3. The
14 metadata may be supplied via the command line, within a cmdlet class definition,
15 and the like. Different components within the administrative tool framework **200**
16 may request the metadata when performing their processing. For example, parser
17 **202** may request metadata to validate parameters supplied on the command line.

18 Exemplary Error & Event Processor

19 The error & event processor **408** provides an error object to store
20 information about each occurrence of an error during processing of a command
21 line. For additional information about one particular error and event processor
22 which is particularly suited for the present administrative tool framework, refer to
23 U.S. Patent Application No. ____/ U.S. Patent No. ____, entitled "System and
24 Method for Persisting Error Information in a Command Line Environment", which
25

1 is owned by the same assignee as the present invention, and is incorporated here
2 by reference.

3 Exemplary Session Manager

4 The session manager **410** supplies session and state information to other
5 components within the administrative tool framework **200**. The state information
6 managed by the session manager may be accessed by any cmdlet, host, or core
7 engine via programming interfaces. These programming interfaces allow for the
8 creation, modification, and deletion of state information.

9 Exemplary Pipeline Processor and Loader

10 The loader **404** is configured to load each cmdlet in memory in order for
11 the pipeline processor **402** to execute the cmdlet. The pipeline processor **402**
12 includes a cmdlet processor **420** and a cmdlet manager **422**. The cmdlet
13 processor **420** dispatches individual cmdlets. If the cmdlet requires execution on a
14 remote, or a set of remote machines, the cmdlet processor **420** coordinates the
15 execution with the remoting cmdlet **236** shown in FIGURE 2. The cmdlet
16 manager **422** handles the execution of aggregations of cmdlets. The cmdlet
17 manager **422**, the cmdlet processor **420**, and the script engine **222** (FIGURE 2)
18 communicate with each other in order to perform the processing on the input
19 received from the host program **210-214**. The communication may be recursive in
20 nature. For example, if the host program provides a script, the script may invoke
21 the cmdlet manager **422** to execute a cmdlet, which itself may be a script. The
22 script may then be executed by the script engine **222**. One exemplary process
23 flow for the core engine is described in detail below in conjunction with FIGURE
24 14.

25 Exemplary Extended Type Manager

1 As mentioned above, the administrative tool framework provides a set of
2 utilities that allows reflection on objects and allows processing on the reflected
3 objects independent of their (object) type. The administrative tool framework **200**
4 interacts with the component framework on the computing system (component
5 framework **120** in FIGURE 1) to perform this reflection. As one skilled in the art
6 will appreciate, reflection provides the ability to query an object and to obtain a
7 type for the object, and then reflect on various objects and properties associated
8 with that type of object to obtain other objects and/or a desired value.

9 Even though reflection provides the administrative tool framework **200** a
10 considerable amount of information on objects, the inventors appreciated that
11 reflection focuses on the type of object. For example, when a database datatable is
12 reflected upon, the information that is returned is that the datatable has two
13 properties: a column property and a row property. These two properties do not
14 provide sufficient detail regarding the “objects” within the datatable. Similar
15 problems arise when reflection is used on extensible markup language (XML) and
16 other objects.

17 Thus, the inventors conceived of an extended type manager **412** that
18 focuses on the usage of the type. For this extended type manager, the type of
19 object is not important. Instead, the extended type manager is interested in
20 whether the object can be used to obtain required information. Continuing with
21 the above datatable example, the inventors appreciated that knowing that the
22 datatable has a column property and a row property is not particularly interesting,
23 but appreciated that one column contained information of interest. Focusing on
24 the usage, one could associate each row with an “object” and associate each
25 column with a “property” of that “object”. Thus, the extended type manager **412**

1 provides a mechanism to create “objects” from any type of precisely parse-able
2 input. In so doing, the extended type manager 412 supplements the reflection
3 capabilities provided by the component-based framework 120 and extends
4 “reflection” to any type of precisely parse-able input.

5 In overview, the extended type manager is configured to access precisely
6 parse-able input (not shown) and to correlate the precisely parse-able input with a
7 requested data type. The extended type manager 412 then provides the requested
8 information to the requesting component, such as the pipeline processor 402 or
9 parser 220. In the following discussion, precisely parse-able input is defined as
10 input in which properties and values may be discerned. Some exemplary precisely
11 parse-able input include Windows Management Instrumentation (WMI) input,
12 ActiveX Data Objects (ADO) input, eXtensible Markup Language (XML) input,
13 and object input, such as .NET objects. Other precisely parse-able input may
14 include third party data formats.

15 Turning briefly to FIGURE 18, a functional block diagram of an exemplary
16 extended type manager for use within the administrative tool framework is shown.
17 For explanation purposes, the functionality (denoted by the number “3” within a
18 circle) provided by the extended type manager is contrasted with the functionality
19 provided by a traditional tightly bound system (denoted by the number “1” within
20 a circle) and the functionality provided by a reflection system (denoted by the
21 number “2” within a circle). In the traditional tightly bound system, a caller 1802
22 within an application directly accesses the information (e.g., properties P1 and P2,
23 methods M1 and M2) within object A. As mentioned above, the caller 1802 must
24 know, a priori, the properties (e.g., properties P1 and P2) and methods (e.g.,
25

1 methods M1 and M2) provided by object A at compile time. In the reflection
2 system, generic code **1820** (not dependent on any data type) queries a system **1808**
3 that performs reflection **1810** on the requested object and returns the information
4 (e.g., properties P1 and P2, methods M1 and M2) about the object (e.g., object A)
5 to the generic code **1820**. Although not shown in object A, the returned
6 information may include additional information, such as vendor, file, date, and the
7 like. Thus, through reflection, the generic code **1820** obtains at least the same
8 information that the tightly bound system provides. The reflection system also
9 allows the caller **1802** to query the system and get additional information without
10 any a priori knowledge of the parameters.

11 In both the tightly bound systems and the reflection systems, new data
12 types can not be easily incorporated within the operating environment. For
13 example, in a tightly bound system, once the operating environment is delivered,
14 the operating environment can not incorporate new data types because it would
15 have to be rebuilt in order to support them. Likewise, in reflection systems, the
16 metadata for each object class is fixed. Thus, incorporating new data types is not
17 usually done.

18 However, with the present extended type manager new data types can be
19 incorporated into the operating system. With the extended type manager **1822**,
20 generic code **1820** may reflect on a requested object to obtain extended data types
21 (e.g., object A') provided by various external sources, such as a third party objects
22 (e.g., object A' and B), a semantic web **1832**, an ontology service **1834**, and the
23 like. As shown, the third party object may extend an existing object (e.g., object
24 A') or may create an entirely new object (e.g., object B).

1 Each of these external sources may register their unique structure within a
2 type metadata 1840 and may provide code 1842. When an object is queried, the
3 extended type manager reviews the type metadata 1840 to determine whether the
4 object has been registered. If the object is not registered within the type metadata
5 1840, reflection is performed. Otherwise, extended reflection is performed. The
6 code 1842 returns the additional properties and methods associated with the type
7 being reflected upon. For example, if the input type is XML, the code 1842 may
8 include a description file that describes the manner in which the XML is used to
9 create the objects from the XML document. Thus, the type metadata 1840
10 describes how the extended type manager 412 should query various types of
11 precisely parse-able input (e.g., third party objects A' and B, semantic web 1832)
12 to obtain the desired properties for creating an object for that specific input type
13 and the code 1842 provides the instructions to obtain these desired properties. As
14 a result, the extended type manager 412 provides a layer of indirection that allows
15 "reflection" on all types of objects.

16 In addition to providing extended types, the extend type manager 412
17 provides additional query mechanisms, such as a property path mechanism, a key
18 mechanism, a compare mechanism, a conversion mechanism, a globber
19 mechanism, a property set mechanism, a relationship mechanism, and the like.
20 Each of these query mechanisms, described below in the section "Exemplary
21 Extended Type Manager Processing", provides flexibility to system administrators
22 when entering command strings. Various techniques may be used to implement
23 the semantics for the extended type manager. Three techniques are described
24 below. However, those skilled in the art will appreciate that variations of these
25

1 techniques may be used without departing from the scope of the claimed
2 invention.

3 In one technique, a series of classes having static methods (e.g.,
4 getProperty()) may be provided. An object is input into the static method (e.g.,
5 getProperty(object)), and the static method returns a set of results. In another
6 technique, the operating environment envelopes the object with an adapter. Thus,
7 no input is supplied. Each instance of the adapter has a getProperty method that
8 acts upon the enveloped object and returns the properties for the enveloped object.
9 The following is pseudo code illustrating this technique:

```
10  
11 Class Adaptor  
12 {  
13     Object X;  
14     getProperties();  
15 }.
```

16
17 In still another technique, an adaptor class subclasses the object.
18 Traditionally, subclassing occurred before compilation. However, with certain
19 operating environments, subclassing may occur dynamically. For these types of
20 environments, the following is pseudo code illustrating this technique:

```
21  
22 Class Adaptor : A  
23 {  
24     getProperties()  
25     {
```

```
1         return data;
2     }
3 }.
```

4 Thus, as illustrated in FIGURE 18, the extended type manager allows
5 developers to create a new data type, register the data type, and allow other
6 applications and cmdlets to use the new data type. In contrast, in prior
7 administrative environments, each data type had to be known at compile time so
8 that a property or method associated with an object instantiated from that data type
9 could be directly accessed. Therefore, adding new data types that were supported
10 by the administrative environment was seldom done in the past.

11 Referring back to FIGURE 2, in overview, the administrative tool
12 framework 200 does not rely on the shell for coordinating the execution of
13 commands input by users, but rather, splits the functionality into processing
14 portions (e.g., host-independent components 206) and user interaction portions
15 (e.g., via host cmdlets). In addition, the present administrative tool environment
16 greatly simplifies the programming of administrative tools because the code
17 required for parsing and data validation is no longer included within each
18 command, but is rather provided by components (e.g., parser 220) within the
19 administrative tool framework. The exemplary processing performed within the
20 administrative tool framework is described below.

21 Exemplary Operation

22 FIGURES 5-7 graphically illustrate exemplary data structures used within
23 the administrative tool environment. FIGURES 8-17 graphically illustrate
24 exemplary processing flows within the administrative tool environment. One
25 skilled in the art will appreciate that certain processing may be performed by a

1 different component than the component described below without departing from
2 the scope of the present invention. Before describing the processing performed
3 within the components of the administrative tool framework, exemplary data
4 structures used within the administrative tool framework are described.

5 Exemplary Data Structures for Cmdlet Objects

6 FIGURE 5 is an exemplary data structure for specifying a cmdlet suitable
7 for use within the administrative tool framework shown in FIGURE 2. When
8 completed, the cmdlet may be a management cmdlet, a non-management cmdlet, a
9 host cmdlet, a provider cmdlet, or the like. The following discussion describes the
10 creation of a cmdlet with respect to a system administrator's perspective (i.e., a
11 provider cmdlet). However, each type of cmdlet is created in the same manner
12 and operates in a similar manner. A cmdlet may be written in any language, such
13 as C#. In addition, the cmdlet may be written using a scripting language or the
14 like. When the administrative tool environment operates with the .NET
15 Framework, the cmdlet may be a .NET object.

16 The provider cmdlet **500** (hereinafter, referred to as cmdlet **500**) is a public
17 class having a cmdlet class name (e.g., StopProcess **504**). Cmdlet **500** derives
18 from a cmdlet class **506**. An exemplary data structure for a cmdlet class **506** is
19 described below in conjunction with FIGURE 6. Each cmdlet **500** is associated
20 with a command attribute **502** that associates a name (e.g., Stop/Process) with the
21 cmdlet **500**. The name is registered within the administrative tool environment.
22 As will be described below, the parser looks in the cmdlet registry to identify the
23 cmdlet **500** when a command string having the name (e.g., Stop/Process) is
24 supplied as input on a command line or in a script.

1 The cmdlet **500** is associated with a grammar mechanism that defines a
2 grammar for expected input parameters to the cmdlet. The grammar mechanism
3 may be directly or indirectly associated with the cmdlet. For example, the cmdlet
4 **500** illustrates a direct grammar association. In this cmdlet **500**, one or more
5 public parameters (e.g., **ProcessName 510** and **PID 512**) are declared. The
6 declaration of the public parameters drives the parsing of the input objects to the
7 cmdlet **500**. Alternatively, the description of the parameters may appear in an
8 external source, such as an XML document. The description of the parameters in
9 this external source would then drive the parsing of the input objects to the cmdlet.

10 Each public parameter **510**, **512** may have one or more attributes (i.e.,
11 directives) associated with it. The directives may be from any of the following
12 categories: parsing directive **521**, data validation directive **522**, data generation
13 directive **523**, processing directive **524**, encoding directive **525**, and
14 documentation directive **526**. The directives may be surrounded by square
15 brackets. Each directive describes an operation to be performed on the following
16 expected input parameter. Some of the directives may also be applied at a class
17 level, such as user-interaction type directives. The directives are stored in the
18 metadata associated with the cmdlet. The application of these attributes is
19 described below in conjunction with FIGURE 12.

20 These attributes may also affect the population of the parameters declared
21 within the cmdlet. One exemplary process for populating these parameters is
22 described below in conjunction with FIGURE 16. The core engine may apply
23 these directives to ensure compliance. The cmdlet **500** includes a first method **530**
24 (hereinafter, interchangeably referred to as **StartProcessing method 530**) and a
25

1 second method **540** (hereinafter, interchangeably referred to as processRecord
2 method **540**). The core engine uses the first and second methods **530**, **540** to
3 direct the processing of the cmdlet **500**. For example, the first method **530** is
4 executed once and performs set-up functions. The code **542** within the second
5 method **540** is executed for each object (e.g., record) that needs to be processed by
6 the cmdlet **500**. The cmdlet **500** may also include a third method (not shown) that
7 cleans up after the cmdlet **500**.

8 Thus, as shown in FIGURE 5, code **542** within the second method **540** is
9 typically quite brief and does not contain functionality required in traditional
10 administrative tool environments, such as parsing code, data validation code, and
11 the like. Thus, system administrators can develop complex administrative tasks
12 without learning a complex programming language.

13 FIGURE 6 is an exemplary data structure **600** for specifying a cmdlet base
14 class **602** from which the cmdlet shown in FIGURE 5 is derived. The cmdlet base
15 class **602** includes instructions that provide additional functionality whenever the
16 cmdlet includes a hook statement and a corresponding switch is input on the
17 command line or in the script (jointly referred to as command input).

18 The exemplary data structure **600** includes parameters, such as Boolean
19 parameter verbose **610**, whatif **620**, and confirm **630**. As will be explained below,
20 these parameters correspond to strings that may be entered on the command input.
21 The exemplary data structure **600** may also include a security method **640** that
22 determines whether the task being requested for execution is allowed.

23 FIGURE 7 is another exemplary data structure **700** for specifying a cmdlet.
24 In overview, the data structure **700** provides a means for clearly expressing a
25

1 contract between the administrative tool framework and the cmdlet. Similar to
2 data structure 500, data structure 700 is a public class that derives from a cmdlet
3 class 704. The software developer specifies a cmdletDeclaration 702 that
4 associates a noun/verb pair, such as "get/process" and "format/table", with the
5 cmdlet 700. The noun/verb pair is registered within the administrative tool
6 environment. The verb or the noun may be implicit in the cmdlet name. Also,
7 similar to data structure 500, data structure 700 may include one or more public
8 members (e.g., Name 730, Recurse 732), which may be associated with the one or
9 more directives 520-526 described in conjunction with data structure 500.

10 However, in this exemplary data structure 700, each of the expected input
11 parameters 730 and 732 is associated with an input attribute 731 and 733,
12 respectively. The input attributes 731 and 733 specifying that the data for its
13 respective parameter 730 and 732 should be obtained from the command line.
14 Thus, in this exemplary data structure 700, there are not any expected input
15 parameters that are populated from a pipelined object that has been emitted by
16 another cmdlet. Thus, data structure 700 does not override the first method (e.g.,
17 StartProcessing) or the second method (e.g., ProcessRecord) which are provided
18 by the cmdlet base class.

19 The data structure 700 may also include a private member 740 that is not
20 recognized as an input parameter. The private member 740 may be used for
21 storing data that is generated based on one of the directives.

22 Thus, as illustrated in data structure 700, through the use of declaring
23 public properties and directives within a specific cmdlet class, cmdlet developers
24 can easily specify a grammar for the expected input parameters to their cmdlets
25

1 and specify processing that should be performed on the expected input parameters
2 without requiring the cmdlet developers to generate any of the underlying logic.
3 Data structure **700** illustrates a direct association between the cmdlet and the
4 grammar mechanism. As mentioned above, this associated may also be indirect,
5 such as by specifying the expected parameter definitions within an external source,
6 such as an XML document.

7 The exemplary process flows within the administrative tool environment
8 are now described.

9 **Exemplary Host Processing Flow**

10 FIGURE 8 is a logical flow diagram illustrating an exemplary process for
11 host processing that is performed within the administrative tool framework shown
12 in FIGURE 2. The process **800** begins at block **801**, where a request has been
13 received to initiate the administrative tool environment for a specific application.
14 The request may have been sent locally through keyboard input, such as selecting
15 an application icon, or remotely through the web services interface of a different
16 computing device. For either scenario, processing continues to block **802**.

17 At block **802**, the specific application (e.g., host program) on the “target”
18 computing device sets up its environment. This includes determining which
19 subsets of cmdlets (e.g., management cmdlets **232**, non-management cmdlets **234**,
20 and host cmdlets **218**) are made available to the user. Typically, the host program
21 will make all the non-management cmdlets **234** available and its own host cmdlets
22 **218** available. In addition, the host program will make a subset of the
23 management cmdlets **234** available, such as cmdlets dealing with processes, disk,
24 and the like. Thus, once the host program makes the subsets of cmdlets available,
25

1 the administrative tool framework is effectively embedded within the
2 corresponding application. Processing continues to block **804**.

3 At block **804**, input is obtained through the specific application. As
4 mentioned above, input may take several forms, such as command lines, scripts,
5 voice, GUI, and the like. For example, when input is obtained via a command
6 line, the input is retrieve from the keystrokes entered on a keyboard. For a GUI
7 host, a string is composed based on the GUI. Processing continues at block **806**.

8 At block **806**, the input is provided to other components within the
9 administrative tool framework for processing. The host program may forward the
10 input directly to the other components, such as the parser. Alternatively, the host
11 program may forward the input via one of its host cmdlets. The host cmdlet may
12 convert its specific type of input (e.g., voice) into a type of input (e.g., text string,
13 script) that is recognized by the administrative tool framework. For example,
14 voice input may be converted to a script or command line string depending on the
15 content of the voice input. Because each host program is responsible for
16 converting their type of input to an input recognized by the administrative tool
17 framework, the administrative tool framework can accept input from any number
18 of various host components. In addition, the administrative tool framework
19 provides a rich set of utilities that perform conversions between data types when
20 the input is forwarded via one of its cmdlets. Processing performed on the input
21 by the other components is described below in conjunction with several other
22 figures. Host processing continues at decision block **808**.

23 At decision block **808**, a determination is made whether a request was
24 received for additional input. This may occur if one of the other components
25 responsible for processing the input needs additional information from the user in

1 order to complete its processing. For example, a password may be required to
2 access certain data, confirmation of specific actions may be needed, and the like.
3 For certain types of host programs (e.g., voice mail), a request such as this may
4 not be appropriate. Thus, instead of querying the user for additional information,
5 the host program may serialize the state, suspend the state, and send a notification
6 so that at a later time the state may be resumed and the execution of the input be
7 continued. In another variation, the host program may provide a default value
8 after a predetermined time period. If a request for additional input is received,
9 processing loops back to block **804**, where the additional input is obtained.
10 Processing then continues through blocks **806** and **808** as described above. If no
11 request for additional input is received and the input has been processed,
12 processing continues to block **810**.

13 At block **810**, results are received from other components within the
14 administrative tool framework. The results may include error messages, status,
15 and the like. The results are in an object form, which is recognized and processed
16 by the host cmdlet within the administrative tool framework. As will be described
17 below, the code written for each host cmdlet is very minimal. Thus, a rich set of
18 output may be displayed without requiring a huge investment in development
19 costs. Processing continues at block **812**.

20 At block **812**, the results may be viewed. The host cmdlet converts the
21 results to the display style supported by the host program. For example, a returned
22 object may be displayed by a GUI host program using a graphical depiction, such
23 as an icon, barking dog, and the like. The host cmdlet provides a default format
24 and output for the data. The default format and output may utilize the exemplary
25

1 output processing cmdlets described below in conjunction with FIGURES 19-23.

2 After the results are optionally displayed, the host processing is complete.

3 **Exemplary Process Flows for Handling Input**

4 FIGURE 9 is a logical flow diagram illustrating an exemplary process for
5 handling input that is performed within the administrative tool framework shown
6 in FIGURE 2. Processing begins at block 901 where input has been entered via a
7 host program and forwarded to other components within the administrative tool
8 framework. Processing continues at block 902.

9 At block 902, the input is received from the host program. In one
10 exemplary administrative tool framework, the input is received by the parser,
11 which deciphers the input and directs the input for further processing. Processing
12 continues at decision block 904.

13 At decision block 904, a determination is made whether the input is a
14 script. The input may take the form of a script or a string representing a command
15 line (hereinafter, referred to as a “command string”). The command string may
16 represent one or more cmdlets pipelined together. Even though the administrative
17 tool framework supports several different hosts, each host provides the input as
18 either a script or a command string for processing. As will be shown below, the
19 interaction between scripts and command strings is recursive in nature. For
20 example, a script may have a line that invokes a cmdlet. The cmdlet itself may be
21 a script.

22 Thus, at decision block 904, if the input is in a form of a script, processing
23 continues at block 906, where processing of the script is performed. Otherwise,
24 processing continues at block 908, where processing of the command string is
25

1 performed. Once the processing performed within either block 906 or 908 is
2 completed, processing of the input is complete.

3 **Exemplary Processing of Scripts**

4 FIGURE 10 is a logical flow diagram illustrating a process for processing a
5 script suitable for use within the process for handling input shown in FIGURE 9.
6 The process begins at block 1001, where the input has been identified as a script.
7 The script engine and parser communicate with each other to perform the
8 following functions. Processing continues at block 1002.

9 At block 1002, pre-processing is performed on the script. Briefly, turning
10 to FIGURE 11, a logical flow diagram is shown that illustrates a script pre-
11 processing process 1100 suitable for use within the script processing process 1000.
12 Script pre-processing begins at block 1101 and continues to decision block 1102.

13 At decision block 1102, a determination is made whether the script is being
14 run for the first time. This determination may be based on information obtained
15 from a registry or other storage mechanism. The script is identified from within
16 the storage mechanism and the associated data is reviewed. If the script has not
17 run previously, processing continues at block 1104.

18 At block 1104, the script is registered in the registry. This allows
19 information about the script to be stored for later access by components within the
20 administrative tool framework. Processing continues at block 1106.

21 At block 1106, help and documentation are extracted from the script and
22 stored in the registry. Again, this information may be later accessed by
23 components within the administrative tool framework. The script is now ready for
24 processing and returns to block 1004 in FIGURE 10.
25

1 Returning to decision block **1102**, if the process concludes that the script
2 has run previously, processing continues to decision block **1108**. At decision block
3 **1108**, a determination is made whether the script failed during processing. This
4 information may be obtained from the registry. If the script has not failed, the
5 script is ready for processing and returns to block **1004** in FIGURE 10.

6 However, if the script has failed, processing continues at block **1110**. At
7 block **1110**, the script engine may notify the user through the host program that the
8 script has previously failed. This notification will allow a user to decide whether
9 to proceed with the script or to exit the script. As mentioned above in conjunction
10 with FIGURE 8, the host program may handle this request in various ways
11 depending on the style of input (e.g., voice, command line). Once additional input
12 is received from the user, the script either returns to block **1004** in FIGURE 10 for
13 processing or the script is aborted.

14 Returning to block **1004** in FIGURE 10, a line from the script is retrieved.
15 Processing continues at decision block **1006**. At decision block **1006**, a
16 determination is made whether the line includes any constraints. A constraint is
17 detected by a predefined begin character (e.g., a bracket “[”) and a corresponding
18 end character (e.g., a close bracket “]”). If the line includes constraints, processing
19 continues to block **1008**.

20 At block **1008**, the constraints included in the line are applied. In general,
21 the constraints provide a mechanism within the administrative tool framework to
22 specify a type for a parameter entered in the script and to specify validation logic
23 which should be performed on the parameter. The constraints are not only
24 applicable to parameters, but are also applicable to any type of construct entered in
25 the script, such as variables. Thus, the constraints provide a mechanism within an

1 interpretive environment to specify a data type and to validate parameters. In
2 traditional environments, system administrators are unable to formally test
3 parameters entered within a script. An exemplary process for applying constraints
4 is illustrated in FIGURE 12.

5 At decision block **1010**, a determination is made whether the line from the
6 script includes built-in capabilities. Built-in capabilities are capabilities that are
7 not performed by the core engine. Built-in capabilities may be processed using
8 cmdlets or may be processed using other mechanisms, such as in-line functions. If
9 the line does not have built-in capabilities, processing continues at decision block
10 **1014**. Otherwise, processing continues at block **1012**.

11 At block **1012**, the built-in capabilities provided on the line of the script are
12 processed. Example built-in capabilities may include execution of control
13 structures, such as “if” statements, “for” loops, switches, and the like. Built-in
14 capabilities may also include assignment type statements (e.g., a=3). Once the
15 built-in capabilities have been processed, processing continues to decision block
16 **1014**.

17 At decision block **1014**, a determination is made whether the line of the
18 script includes a command string. The determination is based on whether the data
19 on the line is associated with a command string that has been registered and with a
20 syntax of the potential cmdlet invocation. As mentioned above, the processing of
21 command strings and scripts may be recursive in nature because scripts may
22 include command strings and command strings may execute a cmdlet that is a
23 script itself. If the line does not include a command string, processing continues at
24 decision block **1018**. Otherwise, processing continues at block **1016**.

1 At block **1016**, the command string is processed. In overview, the
2 processing of the command string includes identifying a cmdlet class by the parser
3 and passing the corresponding cmdlet object to the core engine for execution. The
4 command string may also include a pipelined command string that is parsed into
5 several individual cmdlet objects and individually processed by the core engine.
6 One exemplary process for processing command strings is described below in
7 conjunction with FIGURE 14. Once the command string is processed, processing
8 continues at decision block **1018**.

9 At decision block **1018**, a determination is made whether there is another
10 line in the script. If there is another line in the script, processing loops back to
11 block **1004** and proceeds as described above in blocks **1004-1016**. Otherwise,
12 processing is complete.

13 An exemplary process for applying constraints in block **1008** is illustrated
14 in FIGURE 12. The process begins at block **1201** where a constraint is detected in
15 the script or in the command string on the command line. When the constraint is
16 within a script, the constraints and the associated construct may occur on the same
17 line or on separate lines. When the constraint is within a command string, the
18 constraint and the associated construct occur before the end of line indicator (e.g.,
19 enter key). Processing continues to block **1202**.

20 At block **1202**, constraints are obtained from the interpretive environment.
21 In one exemplary administrative tool environment, the parser deciphers the input
22 and determines the occurrence of constraints. Constraints may be from one of the
23 following categories: predicate directive, parsing directive, data validation
24 directive, data generation directive, processing directive, encoding directive, and
25 documentation directive. In one exemplary parsing syntax, the directives are

1 surrounded by square brackets and describe the construct that follows them. The
2 construct may be a function, a variable, a script, or the like.

3 As will be described below, through the use of directives, script authors are
4 allowed to easily type and perform processing on the parameters within the script
5 or command line (i.e., an interpretive environment) without requiring the script
6 authors to generate any of the underlying logic. Processing continues to block
7 **1204**.

8 At block **1204**, the constraints that are obtained are stored in the metadata
9 for the associated construct. The associated construct is identified as being the
10 first non-attribution token after one or more attribution tokens (tokens that denote
11 constraints) have been encountered. Processing continues to block **1206**.

12 At block **1206**, whenever the construct is encountered within the script or in
13 the command string, the constraints defined within the metadata are applied to the
14 construct. The constraints may include data type, predicate directives **1210**,
15 documentation directives **1212**, parsing directives **1214**, data generation directives
16 **1216**, data validation directives **1218**, and object processing and encoding
17 directives **1220**. Constraints specifying data types may specify any data type
18 supported by the system on which the administrative tool framework is running.
19 Predicate directives **1210** are directives that indicate whether processing should
20 occur. Thus, predicate directives **1210** ensure that the environment is correct for
21 execution. For example, a script may include the following predicate directive:
22

23 [PredicateScript("isInstalled","ApplicationZ")].

24 The predicate directive ensures that the correct application is installed on
25 the computing device before running the script. Typically, system environment

1 variables may be specified as predicate directives. Exemplary directives from
2 directive types 1212-1220 are illustrated in Tables 1-5. Processing of the script is
3 then complete.

4 Thus, the present process for applying types and constraints within an
5 interpretive environment, allows system administrators to easily specify a type,
6 specify validation requirements, and the like without having to write the
7 underlying logic for performing this processing. The following is an example of
8 the constraint processing performed on a command string specified as follows:

9 [Integer][ValidationRange(3,5)]\$a=4.
10

11 There are two constraints specified via attribution tokens denoted by “[]”.
12 The first attribution token indicates that the variable is a type integer and a second
13 attribution token indicates that the value of the variable \$a must be between 3 and
14 5 inclusive. The example command string ensures that if the variable \$a is
15 assigned in a subsequent command string or line, the variable \$a will be checked
16 against the two constraints. Thus, the following command strings would each
17 result in an error:

18 \$a = 231

19 \$a = “apple”

20 \$a = \$(get/location).
21

22 The constraints are applied at various stages within the administrative tool
23 framework. For example, applicability directives, documentation directives, and
24 parsing guideline directives are processed at a very early stage within the parser.
25

Data generation directives and validation directives are processed in the engine once the parser has finished parsing all the input parameters.

The following tables illustrate representative directives for the various categories, along with an explanation of the processing performed by the administrative tool environment in response to the directive.

Name	Description
PrerequisiteMachineRoleAttribute	Informs shell whether element is to be used only in certain machine roles (e.g., File Server, Mail Server).
PrerequisiteUserRoleAttribute	Informs shell whether element is to be used only in certain user roles (e.g., Domain Administrator, Backup Operator).
PrerequisiteScriptAttribute	Informs the shell this script will be run before excuting the actual command or parameter. Can be used for parameter validation
PrerequisiteUITypeAttribute	This is used to check the User interface available before excuting

Table 1. Applicability Directives

1	Name	Description
2	ParsingParameterPositionAttribute	Maps unqualified
3		parameters based on
4		position.
5	ParsingVariableLengthParameterListAttribute	Maps parameters
6		not having a Parsing
7		ParameterPosition
8		attribute.
9	ParsingDisallowInteractionAttribute	Specifies action
10		when number of
11		parameters is less than
12		required number.
13	ParsingRequireInteractionAttribute	Specifies that
14		parameters are obtained
15		through interaction.
16	ParsingHiddenElementAttribute	Makes parameter
17		invisible to end user.
18	ParsingMandatoryParameterAttribute	Specifies that the
19		parameter is required.
20	ParsingPasswordParameterAttribute	Requires special
21		handling of parameter.
22	ParsingPromptStringAttribute	Specifies a prompt
23		for the parameter.
24	ParsingDefaultAnswerAttribute	Specifies default
25		

		answer for parameter.
ParsingDefaultAnswerScriptAttribute		Specifies action to get default answer for parameter.
ParsingDefaultValueAttribute		Specifies default value for parameter.
ParsingDefaultValueScriptAttribute		Specifies action to get default value for parameter.
ParsingParameterMappingAttribute		Specifies a way to group parameters
ParsingParameterDeclarationAttribute		This defines that the filed is a parameter
ParsingAllowPipelineInputAttribute		Defines the parameter can be populated from the pipeline

Table 2. Parsing Guideline Directives

Name	Description
DocumentNameAttribute	Provides a Name to refer to elements for interaction or help.
DocumentShortDescriptionAttribute	Provides brief description of

	element.
DocumentLongDescriptionAttribute	Provides detailed description of element.
DocumentExampleAttribute	Provides example of element.
DocumentSeeAlsoAttribute	Provides a list of related elements.
DocumentSynopsisAttribute	Provides documentation information for element.

Table 3. Documentation Directives

Name	Description
ValidationRangeAttribute	Specifies that parameter must be within certain range.
ValidationSetAttribute	Specifies that parameter must be within certain collection.
ValidationPatternAttribute	Specifies that parameter must fit a certain pattern.
ValidationLengthAttribute	Specifies the strings must be within size range.
ValidationTypeAttribute	Specifies that parameter must be of certain type.
ValidationCountAttribute	Specifies that input items must

	be of a certain number.
ValidationFileAttribute	Specifies certain properties for a file.
ValidationFileAttributesAttribute	Specifies certain properties for a file.
ValidationFileSizeAttribute	Specifies that files must be within specified range.
ValidationNetworkAttribute	Specifies that given Network Entity supports certain properties.
ValidationScriptAttribute	Specifies conditions to evaluate before using element.
ValidationMethodAttribute	Specifies conditions to evaluate before using element.

Table 4. Data Validation Directives

Name	Description
ProcessingTrimStringAttribute	Specifies size limit for strings.
ProcessingTrimCollectionAttribute	Specifies size limit for collection.
EncodingTypeCoercionAttribute	Specifies Type that objects are to be encoded.
ExpansionWildcardsAttribute	Provides a mechanism to allow

Table 5. Processing and Encoding Directives

When the exemplary administrative tool framework is operating within the .NET™ Framework, each category has a base class that is derived from a basic category class (e.g., CmdAttribute). The basic category class derives from a System.Attribute class. Each category has a pre-defined function (e.g., attrib.func()) that is called by the parser during category processing. The script author may create a custom category that is derived from a custom category class (e.g., CmdCustomAttribute). The script author may also extend an existing category class by deriving a directive class from the base category class for that category and override the pre-defined function with their implementation. The script author may also override directives and add new directives to the pre-defined set of directives.

The order of processing of these directives may be stored in an external data store accessible by the parser. The administrative tool framework looks for registered categories and calls a function (e.g., ProcessCustomDirective) for each of the directives in that category. Thus, the order of category processing may be dynamic by storing the category execution information in a persistent store. At different processing stages, the parser checks in the persistent store to determine if any metadata category needs to be executed at that time. This allows categories to be easily deprecated by removing the category entry from the persistent store.

Exemplary Processing of Command Strings

One exemplary process for processing command strings is now described. FIGURE 13 is a functional flow diagram graphically illustrating the processing of a command string 1350 through a parser 220 and a core engine 224 within the administrative tool framework shown in FIGURE 2. The exemplary command string 1350 pipelines several commands (i.e., process command 1360, where command 1362, sort command 1364, and table command 1366). The command line 1350 may pass input parameters to any of the commands (e.g., "handlecount > 400" is passed to the where command 1362). One will note that the process command 1360 does not have any associated input parameters.

In the past, each command was responsible for parsing the input parameters associated with the command, determining whether the input parameters were valid, and issuing error messages if the input parameters were not valid. Because the commands were typically written by various programmers, the syntax for the input parameters on the command line was not very consistent. In addition, if an error occurred, the error message, even for the same error, was not very consistent between the commands.

For example, in a UNIX environment, an "ls" command and a "ps" command have many inconsistencies between them. While both accept an option "-w", the "-w" option is used by the "ls" command to denote the width of the page, while the "-w" option is used by the "ps" command to denote print wide output (in essence, ignoring page width). The help pages associated with the "ls" and the "ps" command have several inconsistencies too, such as having options bolded in

1 one and not the other, sorting options alphabetically in one and not the other,
2 requiring some options to have dashes and some not.

3 The present administrative tool framework provides a more consistent
4 approach and minimizes the amount of duplicative code that each developer must
5 write. The administrative tool framework 200 provides a syntax (e.g., grammar), a
6 corresponding semantics (e.g., a dictionary), and a reference model to enable
7 developers to easily take advantage of common functionality provided by the
8 administrative tool framework 200.

9 Before describing the present invention any further, definitions for
10 additional terms appearing through-out this specification are provided. Input
11 parameter refers to input-fields for a cmdlet. Argument refers to an input
12 parameter passed to a cmdlet that is the equivalent of a single string in the argv
13 array or passed as a single element in a cmdlet object. As will be described below,
14 a cmdlet provides a mechanism for specifying a grammar. The mechanism may
15 be provided directly or indirectly. An argument is one of an option, an option-
16 argument, or an operand following the command-name. Examples of arguments
17 are given based on the following command line:
18

19
20 findstr /i /d:\winnt;\winnt\system32 aa*b *.ini.
21

22 In the above command line, "findstr" is argument 0, "/i" is argument 1,
23 "/d:\winnt;\winnt\system32" is argument 2, "aa*b" is argument 3, and "*.ini" is
24 argument 4. An "option" is an argument to a cmdlet that is generally used to
25

1 specify changes to the program's default behavior. Continuing with the example
2 command line above, "/i" and "/d" are options. An "option-argument" is an input
3 parameter that follows certain options. In some cases, an option-argument is
4 included within the same argument string as the option. In other cases, the option-
5 argument is included as the next argument. Referring again to the above
6 command line, "winnt;\winnt\system32" is an option-argument. An "operand" is
7 an argument to a cmdlet that is generally used as an object supplying information
8 to a program necessary to complete program processing. Operands generally
9 follow the options in a command line. Referring to the example command line
10 above again, "aa*b" and "*.ini" are operands. A "parsable stream" includes the
11 arguments.

12 Referring to FIGURE 13, parser 220 parses a parsable stream (e.g.,
13 command string 1350) into constituent parts 1320-1326 (e.g., where portion 1322).
14 Each portion 1320-1326 is associated with one of the cmdlets 1330-1336. Parser
15 220 and engine 224 perform various processing, such as parsing, parameter
16 validation, data generation, parameter processing, parameter encoding, and
17 parameter documentation. Because parser 220 and engine 224 perform common
18 functionality on the input parameters on the command line, the administrative tool
19 framework 200 is able to issue consistent error messages to users.

20 As one will recognize, the executable cmdlets 1330-1336 written in
21 accordance with the present administrative tool framework require less code than
22 commands in prior administrative environments. Each executable cmdlet 1330-
23 1336 is identified using its respective constituent part 1320-1326. In addition,
24 each executable cmdlet 1330-1336 outputs objects (represented by arrows 1340,
25

1 1342, 1344, and 1346) which are input as input objects (represented by arrows
2 1341, 1343, and 1345) to the next pipelined cmdlet. These objects may be input
3 by passing a reference (e.g., handle) to the object. The executable cmdlets 1330-
4 1336 may then perform additional processing on the objects that were passed in.

5 FIGURE 14 is a logical flow diagram illustrating in more detail the
6 processing of command strings suitable for use within the process for handling
7 input shown in FIGURE 9. The command string processing begins at block 1401,
8 where either the parser or the script engine identified a command string within the
9 input. In general the core engine performs set-up and sequencing of the data flow
10 of the cmdlets. The set-up and sequencing for one cmdlet is described below, but
11 is applicable to each cmdlet in a pipeline. Processing continues at block 1404.

12 At block 1404, a cmdlet is identified. The identification of the cmdlet may
13 be thru registration. The core engine determines whether the cmdlet is local or
14 remote. The cmdlet may execute in the following locations: 1) within the
15 application domain of the administrative tool framework; 2) within another
16 application domain of the same process as the administrative tool framework; 3)
17 within another process on the same computing device; or 4) within a remote
18 computing device. The communication between cmdlets operating within the
19 same process is through objects. The communication between cmdlets operating
20 within different processes is through a serialized structured data format. One
21 exemplary serialized structured data format is based on the extensible markup
22 language (XML). Processing continues at block 1406.

23 At block 1406, an instance of the cmdlet object is created. An exemplary
24 process for creating an instance of the cmdlet is described below in conjunction
25

1 with FIGURE 15. Once the cmdlet object is created, processing continues at
2 block **1408**.

3 At block **1408**, the properties associated with the cmdlet object are
4 populated. As described above, the developer declares properties within a cmdlet
5 class or within an external source. Briefly, the administrative tool framework will
6 decipher the incoming object(s) to the cmdlet instantiated from the cmdlet class
7 based on the name and type that is declared for the property. If the types are
8 different, the type may be coerced via the extended data type manager. As
9 mentioned earlier, in pipelined command strings, the output of each cmdlet may be
10 a list of handles to objects. The next cmdlet may inputs this list of object handles,
11 performs processing, and passes another list of object handles to the next cmdlet.
12 In addition, as illustrated in FIGURE 7, input parameters may be specified as
13 coming from the command line. One exemplary method for populating properties
14 associated with a cmdlet is described below in conjunction with FIGURE 16.
15 Once the cmdlet is populated, processing continues at block **1410**.

16 At block **1410**, the cmdlet is executed. In overview, the processing
17 provided by the cmdlet is performed at least once, which includes processing for
18 each input object to the cmdlet. Thus, if the cmdlet is the first cmdlet within a
19 pipelined command string, the processing is executed once. For subsequent
20 cmdlets, the processing is executed for each object that is passed to the cmdlet.
21 One exemplary method for executing cmdlets is described below in conjunction
22 with FIGURE 5. When the input parameters are only coming from the command
23 line, execution of the cmdlet uses the default methods provided by the base cmdlet
24 case. Once the cmdlet is finished executing, processing proceeds to block **1412**.

1 At block 1412, the cmdlet is cleaned-up. This includes calling the
2 destructor for the associated cmdlet object which is responsible for de-allocating
3 memory and the like. The processing of the command string is then complete.

4 Exemplary Process for Creating a Cmdlet Object

5 FIGURE 15 is a logical flow diagram illustrating an exemplary process for
6 creating a cmdlet object suitable for use within the processing of command strings
7 shown in FIGURE 14. At this point, the cmdlet data structure has been developed
8 and attributes and expected input parameters have been specified. The cmdlet has
9 been compiled and has been registered. During registration, the class name (i.e.,
10 cmdlet name) is written in the registration store and the metadata associated with
11 the cmdlet has been stored. The process 1500 begins at block 1501, where the
12 parser has received input (e.g., keystrokes) indicating a cmdlet. The parser may
13 recognize the input as a cmdlet by looking up the input from within the registry
14 and associating the input with one of the registered cmdlets. Processing proceeds
15 to block 1504.

16 At block 1504, metadata associated with the cmdlet object class is read.
17 The metadata includes any of the directives associated with the cmdlet. The
18 directives may apply to the cmdlet itself or to one or more of the parameters.
19 During cmdlet registration, the registration code registers the metadata into a
20 persistent store. The metadata may be stored in an XML file in a serialized
21 format, an external database, and the like. Similar to the processing of directives
22 during script processing, each category of directives is processed at a different
23 stage. Each metadata directive handles its own error handling. Processing
24 continues at block 1506.

1 At block **1506**, a cmdlet object is instantiated based on the identified cmdlet
2 class. Processing continues at block **1508**.

3 At block **1508**, information is obtained about the cmdlet. This may occur
4 through reflection or other means. The information is about the expected input
5 parameters. As mentioned above, the parameters that are declared public (e.g.,
6 public string Name 730) correspond to expected input parameters that can be
7 specified in a command string on a command line or provided in an input stream.
8 The administrative tool framework through the extended type manager, described
9 in FIGURE 18, provides a common interface for returning the information (on a
10 need basis) to the caller. Processing continues at block **1510**.

11 At block **1510**, applicability directives (e.g., Table 1) are applied. The
12 applicability directives insure that the class is used in certain machine roles and/or
13 user roles. For example, certain cmdlets may only be used by Domain
14 Administrators. If the constraint specified in one of the applicability directives is
15 not met, an error occurs. Processing continues at block **1512**.

16 At block **1512**, metadata is used to provide intellisense. At this point in
17 processing, the entire command string has not yet been entered. The
18 administrative tool framework, however, knows the available cmdlets. Once a
19 cmdlet has been determined, the administrative tool framework knows the input
20 parameters that are allowed by reflecting on the cmdlet object. Thus, the
21 administrative tool framework may auto-complete the cmdlet once a
22 disambiguating portion of the cmdlet name is provided, and then auto-complete
23 the input parameter once a disambiguating portion of the input parameter has been
24 typed on the command line. Auto-completion may occur as soon as the portion of
25

1 the input parameter can identify one of the input parameters unambiguously. In
2 addition, auto-completion may occur on cmdlet names and operands too.
3 Processing continues at block 1514.

4 At block 1514, the process waits until the input parameters for the cmdlet
5 have been entered. This may occur once the user has indicated the end of the
6 command string, such as by hitting a return key. In a script, a new line indicates
7 the end of the command string. This wait may include obtaining additional
8 information from the user regarding the parameters and applying other directives.
9 When the cmdlet is one of the pipelined parameters, processing may begin
10 immediately. Once, the necessary command string and input parameters have
11 been provided, processing is complete.

12 Exemplary Process for Populating the Cmdlet

13 An exemplary process for populating a cmdlet is illustrated in FIGURE 16
14 and is now described, in conjunction with FIGURE 5. In one exemplary
15 administrative tool framework, the core engine performs the processing to
16 populate the parameters for the cmdlet. Processing begins at block 1601 after an
17 instance of a cmdlet has been created. Processing continues to block 1602.

18 At block 1602, a parameter (e.g., ProcessName) declared within the cmdlet
19 is retrieved. Based on the declaration with the cmdlet, the core engine recognizes
20 that the incoming input objects will provide a property named "ProcessName". If
21 the type of the incoming property is different than the type specified in the
22 parameter declaration, the type will be coerced via the extended type manager.
23 The process of coercing data types is explained below in the subsection entitled
24
25

1 “Exemplary Extended Type Manager Processing.” Processing continues to block
2 1603.

3 At block 1603, an attribute associated with the parameter is obtained. The
4 attribute identifies whether the input source for the parameter is the command line
5 or whether it is from the pipeline. Processing continues to decision block 1604.

6 At decision block 1604, a determination is made whether the attribute
7 specifies the input source as the command line. If the input source is the
8 command line, processing continues at block 1609. Otherwise, processing
9 continues at decision block 1605.

10 At decision block 1605, a determination is made whether the property name
11 specified in the declaration should be used or whether a mapping for the property
12 name should be used. This determination is based on whether the command input
13 specified a mapping for the parameter. The following line illustrates an exemplary
14 mapping of the parameter “ProcessName” to the “foo” member of the incoming
15 object:

16 \$ get/process | where han* -gt 500 | stop/process -ProcessName<-foo.

17 Processing continues at block 1606.

18 At block 1606, the mapping is applied. The mapping replaces the name of
19 the expected parameter from “ProcessName” to “foo”, which is then used by the
20 core engine to parse the incoming objects and to identify the correct expected
21 parameter. Processing continues at block 1608.

22 At block 1608, the extended type manager is queried to locate a value for
23 the parameter within the incoming object. As explain in conjunction with the
24 extended type manager, the extended type manager takes the parameter name and
25 uses reflection to identify a parameter within the incoming object with parameter

1 name. The extended type manager may also perform other processing for the
2 parameter, if necessary. For example, the extended type manager may coerce the
3 type of data to the expected type of data through a conversion mechanism
4 described above. Processing continues to decision block **1610**.

5 Referring back to block 1609, if the attribute specifies that the input source
6 is the command line, data from the command line is obtained. Obtaining the data
7 from the command line may be performed via the extended type manager.
8 Processing then continues to decision block **1610**.

9 At decision block **1610**, a determination is made whether there is another
10 expected parameter. If there is another expected parameter, processing loops back
11 to block **1602** and proceeds as described above. Otherwise, processing is
12 complete and returns.

13 Thus, as shown, cmdlets act as a template for shredding incoming data to
14 obtain the expected parameters. In addition, the expected parameters are obtained
15 without knowing the type of incoming object providing the value for the expected
16 parameter. This is quite different than traditional administrative environments.
17 Traditional administrative environments are tightly bound and require that the type
18 of object be known at compile time. In addition, in traditional environments, the
19 expected parameter would have been passed into the function by value or by
20 reference. Thus, the present parsing (e.g., “shredding”) mechanism allows
21 programmers to specify the type of parameter without requiring them to
22 specifically know how the values for these parameters are obtained.

23 For example, given the following declaration for the cmdlet Foo:
24
25

```

1      class Foo : Cmdlet
2      {
3
4          string Name;
5
6          Bool Recurse;
7      }

```

The command line syntax may be any of the following:

```

11     $ Foo -Name: (string) -Recurse: True
12
13     $ Foo -Name <string> -Recurse True
14
15     $Foo -Name (string).

```

The set of rules may be modified by system administrators in order to yield a desired syntax. In addition, the parser may support multiple sets of rules, so that more than one syntax can be used by users. In essence, the grammar associated with the cmdlet structure (e.g., string Name and Bool Recurse) drives the parser.

In general, the parsing directives describe how the parameters entered as the command string should map to the expected parameters identified in the cmdlet object. The input parameter types are checked to determine whether correct. If the input parameter types are not correct, the input parameters may be coerced to become correct. If the input parameter types are not correct and can not be coerced, a usage error is printed. The usage error allows the user to become

1 aware of the correct syntax that is expected. The usage error may obtain
2 information describing the syntax from the Documentation Directives. Once the
3 input parameter types have either been mapped or have been verified, the
4 corresponding members in the cmdlet object instance are populated. As the
5 members are populated, the extended type manager provides processing of the
6 input parameter types. Briefly, the processing may include a property path
7 mechanism, a key mechanism, a compare mechanism, a conversion mechanism, a
8 globber mechanism, a relationship mechanism, and a property set mechanism.
9 Each of these mechanisms is described in detail below in the section entitled
10 “Extended Type Manager Processing”, which also includes illustrative examples.

11 Exemplary Process for Executing the Cmdlet

12 An exemplary process for executing a cmdlet is illustrated in FIGURE 17
13 and is now described. In one exemplary administrative tool environment, the core
14 engine executes the cmdlet. As mentioned above, the code **1442** within the second
15 method **1440** is executed for each input object. Processing begins at block **1701**
16 where the cmdlet has already been populated. Processing continues at block **1702**.

17 At block **1702**, a statement from the code **542** is retrieved for execution.
18 Processing continues at decision block **1704**.

19 At decision block **1704**, a determination is made whether a hook is included
20 within the statement. Turning briefly to FIGURE 5, the hook may include calling
21 an API provided by the core engine. For example, statement **550** within the code
22 **542** of cmdlet **500** in FIGURE 5 calls the confirmprocessing API specifying the
23 necessary parameters, a first string (e.g., “PID=”), and a parameter (e.g., PID).
24 Turning back to FIGURE 17, if the statement includes the hook, processing
25

1 continues to block **1712**. Thus, if the instruction calling the confirmprocessing
2 API is specified, the cmdlet operates in an alternate executing mode that is
3 provided by the operating environment. Otherwise, processing continues at block
4 **1706** and execution continues in the “normal” mode.

5 At block **1706**, the statement is processed. Processing then proceeds to
6 decision block **1708**. At block **1708**, a determination is made whether the code
7 includes another statement. If there is another statement, processing loops back to
8 block **1702** to get the next statement and proceeds as described above. Otherwise,
9 processing continues to decision block **1714**.

10 At decision block **1714**, a determination is made whether there is another
11 input object to process. If there is another input object, processing continues to
12 block **1716** where the cmdlet is populated with data from the next object. The
13 population process described in FIGURE 16 is performed with the next object.
14 Processing then loops back to block **1702** and proceeds as described above. Once
15 all the objects have been processed, the process for executing the cmdlet is
16 complete and returns.

17 Returning back to decision block **1704**, if the statement includes the hook,
18 processing continues to block **1712**. At block **1712**, the additional features
19 provided by the administrative tool environment are processed. Processing
20 continues at decision block **1708** and continues as described above.

21 The additional processing performed within block **1712** is now described in
22 conjunction with the exemplary data structure **600** illustrated in FIGURE 6. As
23 explained above, within the command base class **600** there may be parameters
24 declared that correspond to additional expected input parameters (e.g., a switch).
25

1 The switch includes a predetermined string, and when recognized, directs
2 the core engine to provide additional functionality to the cmdlet. If the parameter
3 verbose **610** is specified in the command input, verbose statements **614** are
4 executed. The following is an example of a command line that includes the
5 verbose switch:

6
7 \$ get/process | where "han* -gt 500" | stop/process -verbose.

8
9 In general, when "-verbose" is specified within the command input, the
10 core engine executes the command for each input object and forwards the actual
11 command that was executed for each input object to the host program for display.
12 The following is an example of output generated when the above command line is
13 executed in the exemplary administrative tool environment:

14
15 \$ stop/process PID=15

16 \$ stop/process PID=33.

17
18 If the parameter whatif **620** is specified in the command input, whatif
19 statements **624** are executed. The following is an example of a command line that
20 includes the whatif switch:

21
22 \$ get/process | where "han* -gt 500" | stop/process -whatif.

23
24 In general, when "-whatif" is specified, the core engine does not actually
25 execute the code **542**, but rather sends the commands that would have been

1 executed to the host program for display. The following is an example of output
2 generated when the above command line is executed in the administrative tool
3 environment of the present invention:

4
5 #\$ stop/process PID=15

6 #\$ stop/process PID=33.

7
8 If the parameter confirm **630** is specified in the command input, confirm
9 statements **634** are executed. The following is an example of a command line that
10 includes the confirm switch:

11
12 \$ get/process | where "han* -gt 500" | stop/process -confirm.

13
14 In general, when "-confirm" is specified, the core engine requests
15 additional user input on whether to proceed with the command or not. The
16 following is an example of output generated when the above command line is
17 executed in the administrative tool environment of the present invention.

18
19 \$ stop/process PID 15

20 Y/N Y

21 \$ stop/process PID 33

22 Y/N N.

23
24 As described above, the exemplary data structure **600** may also include a
25 security method **640** that determines whether the task being requested for

1 execution should be allowed. In traditional administrative environments, each
2 command is responsible for checking whether the person executing the command
3 has sufficient privileges to perform the command. In order to perform this check,
4 extensive code is needed to access information from several sources. Because of
5 these complexities, many commands did not perform a security check. The
6 inventors of the present administrative tool environment recognized that when the
7 task is specified in the command input, the necessary information for performing
8 the security check is available within the administrative tool environment.
9 Therefore, the administrative tool framework performs the security check without
10 requiring complex code from the tool developers. The security check may be
11 performed for any cmdlet that defines the hook within its cmdlet. Alternatively,
12 the hook may be an optional input parameter that can be specified in the command
13 input, similar to the verbose parameter described above.

14 The security check is implemented to support roles based authentication,
15 which is generally defined as a system of controlling which users have access to
16 resources based on the role of the user. Thus, each role is assigned certain access
17 rights to different resources. A user is then assigned to one or more roles. In
18 general, roles based authentication focus on three items: principle, resource, and
19 action. The *principle* identifies who requested the *action* to be performed on the
20 *resource*.

21 The inventors of the present invention recognized that the cmdlet being
22 requested corresponded to the action that was to be performed. In addition, the
23 inventors appreciated that the owner of the process in which the administrative
24 tool framework was executing corresponded to the principle. Further, the
25 inventors appreciated that the resource is specified within the cmdlet. Therefore,

1 because the administrative tool framework has access to these items, the inventors
2 recognized that the security check could be performed from within the
3 administrative tool framework without requiring tool developers to implement the
4 security check.

5 The operation of the security check may be performed any time additional
6 functionality is requested within the cmdlet by using the hook, such as the
7 confirmprocessing API. Alternatively, security check may be performed by
8 checking whether a security switch was entered on the command line, similar to
9 verbose, whatif, and confirm. For either implementation, the checkSecurity
10 method calls an API provided by a security process (not shown) that provides a set
11 of APIs for determining who is allowed. The security process takes the
12 information provided by the administrative tool framework and provides a result
13 indicating whether the task may be completed. The administrative tool framework
14 may then provide an error or just stop the execution of the task.

15 Thus, by providing the hook within the cmdlet, the developers may use
16 additional processing provided by the administrative tool framework.

17 Exemplary Extended Type Manager Processing

18 As briefly mentioned above in conjunction with FIGURE 18, the extended
19 type manager may perform additional processing on objects that are supplied. The
20 additional processing may be performed at the request of the parser 220, the script
21 engine 222, or the pipeline processor 402. The additional processing includes a
22 property path mechanism, a key mechanism, a compare mechanism, a conversion
23 mechanism, a globber mechanism, a relationship mechanism, and a property set
24 mechanism. Those skilled in the art will appreciate that the extended type
25 manager may also be extended with other processing without departing from the

1 scope of the claimed invention. Each of the additional processing mechanisms is
2 now described.

3 First, the property path mechanism allows a string to navigate properties of
4 objects. In current reflection systems, queries may query properties of an object.
5 However, in the present extended type manager, a string may be specified that will
6 provide a navigation path to successive properties of objects. The following is an
7 illustrative syntax for the property path: P1.P2.P3.P4.

8 Each component (e.g., P1, P2, P3, and P4) comprises a string that may
9 represent a property, a method with parameters, a method without parameters, a
10 field, an XPATH, or the like. An XPATH specifies a query string to search for an
11 element (e.g., “/FOO@=13”). Within the string, a special character may be
12 included to specifically indicate the type of component. If the string does not
13 contain the special character, the extended type manager may perform a lookup to
14 determine the type of component. For example, if component P1 is an object, the
15 extended type manager may query whether P2 is a property of the object, a
16 method on the object, a field of the object, or a property set. Once the extended
17 type manager identifies the type for P2, processing according to that type is
18 performed. If the component is not one of the above types, the extended type
19 manager may further query the extended sources to determine whether there is a
20 conversion function to convert the type of P1 into the type of P2. These and other
21 lookups will now be described using illustrative command strings and showing the
22 respective output.

23 The following is an illustrative string that includes a property path:
24
25

1 \$ get/process | /where hand* -gt 500 | format/table name.toupper, ws.kb,
2 exe*.ver*.description.tolower.trunc(30).

3 In the above illustrative string, there are three property paths: (1)
4 “name.toupper”; (2) “ws.kb”; and (3) “exe*.ver*.description.tolower.trunc(30).
5 Before describing these property paths, one should note that “name”, “ws”, and
6 “exe” specify the properties for the table. In addition, one should note that each of
7 these properties is a direct property of the incoming object, originally generated by
8 “get/process” and then pipelined through the various cmdlets. Processing
9 involved for each of the three property paths will now be described.

10 In the first property path (i.e., “name.toupper”), name is a direct property of
11 the incoming object and is also an object itself. The extended type manager
12 queries the system using the priority lookup described above to determine the type
13 for toupper. The extended type manager discovers that toupper is not a property.
14 However, toupper may be a method inherited by a string type to convert lower
15 case letters to upper case letters within the string. Alternatively, the extended type
16 manager may have queried the extended metadata to determine whether there is
17 any third party code that can convert a name object to upper case. Upon finding
18 the component type, processing is performed in accordance with that component
19 type.

20 In the second property path (i.e., “ws.kb”), “ws” is a direct property of the
21 incoming object and is also an object itself. The extended type manager
22 determines that “ws” is an integer. Then, the extended type manager queries
23 whether kb is a property of an integer, whether kb is a method of an integer, and
24 finally queries whether any code knows how to take an integer and convert the
25

integer to a kb type. Third party code is registered to perform this conversion and the conversion is performed.

In the third property path (i.e., "exe*.ver*.description.tolower.trunc(30)"), there are several components. The first component ("exe*") is a direct property of the incoming object and is also an object. Again, the extended type manager proceeds down the lookup query in order to process the second component ("ver*"). The "exe*" object does not have a "ver*" property or method, so the extend type manager queries the extended metadata to determine whether there is any code that is registered to convert an executable name into a version. For this example, such code exists. The code may take the executable name string and use it to open a file, then accesses the version block object, and return the description property (the third component ("description") of the version block object. The extended type manager then performs this same lookup mechanism for the fourth component ("tolower") and the fifth component ("trunc(40)"). Thus, as illustrated, the extended type manager may perform quite elaborate processing on a command string without the administrator needing to write any specific code. Table 1 illustrates output generated for the illustrative string.

Name.toupper ws.kb exe*.ver*.description.tolower.trunc(30)

ETCLIENT 29,964 etclient

CSRSS 6,944

SVCHOST 28,944 generic host process for win32

OUTLOOK 18,556 office outlook

MSMSGs 13,248 messenger

Table 1.

Another query mechanism 1824 includes a key. The key identifies one or more properties that make an instance of the data type unique. For example, in a database, one column may be identified as the key which can uniquely identify each row (e.g., social security number). The key is stored within the type metadata 1840 associated with the data type. This key may then be used by the extended type manager when processing objects of that data type. The data type may be an extended data type or an existing data type.

Another query mechanism 1824 includes a compare mechanism. The compare mechanism compares two objects. If the two objects directly support the compare function, the directly supported compare function is executed. However, if neither object supports a compare function, the extended type manager may look in the type metadata for code that has been registered to support the compare between the two objects. An illustrative series of command line strings invoking the compare mechanism is shown below, along with corresponding output in Table 2.

```
$ $a = $( get/date )
```

```
$ start/sleep 5
```

```
$ $b = $( get/date
```

```
compare/time $a $b
```

```
Ticks      : 51196579
```

```
Days       : 0
```

```
Hours      : 0
```

```
Milliseconds : 119
```

```
Minutes    : 0
```

1 Seconds : 5
2 TotalDays : 5.92552997685185E-05
3 TotalHours : 0.00142212719444444
4 TotalMilliseconds : 5119.6579
5 TotalMinutes : 0.0853276316666667
6 TotalSeconds : 5.1196579

7 Table 2.

8 Compare/time cmdlet is written to compare two datetime objects. In this
9 case, the DateTime object supports the IComparable interface.

10
11 Another query mechanism **1824** includes a conversion mechanism. The
12 extended type manager allows code to be registered stating its ability to perform a
13 specific conversion. Then, when an object of type A is input and a cmdlet
14 specifies an object of type B, the extended type manager may perform the
15 conversion using one of the registered conversions. The extended type manager
16 may perform a series of conversions in order to coerce type A into type B. The
17 property path described above (“ws.kb”) illustrates a conversion mechanism.

18 Another query mechanism **1824** includes a globber mechanism. A globber
19 refers to a wild card character within a string. The globber mechanism inputs the
20 string with the wild card character and produces a set of objects. The extended
21 type manager allows code to be registered that specifies wildcard processing. The
22 property path described above (“exe*.ver*.description.tolower.trunc(30))
23 illustrates the globber mechanism. A registered process may provide globbing for
24 file names, file objects, incoming properties, and the like.
25

Another query mechanism 1824 includes a property set mechanism. The property set mechanism allows a name to be defined for a set of properties. An administrator may then specify the name within the command string to obtain the set of properties. The property set may be defined in various ways. In one way, a predefined parameter, such as “?”, may be entered as an input parameter for a cmdlet. The operating environment upon recognizing the predefined parameter lists all the properties of the incoming object. The list may be a GUI that allows an administrator to easily check (e.g., “click on”) the properties desired and name the property set. The property set information is then stored in the extended metadata. An illustrative string invoking the property set mechanism is shown below, along with corresponding output in Table 3:

\$ get/process | where han* -gt 500 | format/table config.

In this illustrative string, a property set named “config” has been defined to include a name property, a process id property (Pid), and a priority property. The output for the table is shown below.

<u>Name</u>	<u>Pid</u>	<u>Priority</u>
ETClient	3528	Normal
csrss	528	Normal
svchost	848	Normal
OUTLOOK	2,772	Normal
msmsgs	2,584	Normal

Table 3.

1 Another query mechanism 1824 includes a relationship mechanism. In
2 contrast to traditional type systems that support one relationship (i.e., inheritance),
3 the relationship mechanism supports expressing more than one relationship
4 between types. Again, these relationships are registered. The relationship may
5 include finding items that the object consumes or finding the items that consume
6 the object. The extended type manager may access ontologies that describe
7 various relationships. Using the extended metadata and the code, a specification
8 for accessing any ontology service, such as OWL, DAWL, and the like, may be
9 described. The following is a portion of an illustrative string which utilizes the
10 relationship mechanism: .OWL:"string".

11 The "OWL" identifier identifies the ontology service and the "string"
12 specifies the specific string within the ontology service. Thus, the extended type
13 manager may access types supplied by ontology services.

14 Exemplary Process for Displaying Command Line Data

15 The present mechanism provides a data driven command line output. The
16 formatting and outputting of the data is provided by one or more cmdlets in the
17 pipeline of cmdlets. Typically, these cmdlets are included within the non-
18 management cmdlets described in conjunction with FIGURE 2 above. The
19 cmdlets may include a format cmdlet, a markup cmdlet, a convert cmdlet, a
20 transform cmdlet, and an out cmdlet.

21 FIGURE 19 graphically depicts exemplary sequences 1901-1907 of these
22 cmdlets within a pipeline. The first sequence 1901 illustrates the out cmdlet 1910
23 as the last cmdlet in the pipeline. In the same manner as described above for other
24 cmdlets, the out cmdlet 1910 accepts a stream of pipeline objects generated and
25

1 processed by other cmdlets within the pipeline. However, in contrast to most
2 cmdlets, the out cmdlet **1910** does not emit pipeline objects for other cmdlets.
3 Instead, the out cmdlet **1910** is responsible for rendering/displaying the results
4 generated by the pipeline. Each out cmdlet **1910** is associated with an output
5 destination, such as a device, a program, and the like. For example, for a console
6 device, the out cmdlet **1910** may be specified as out/console; for an internet
7 browser, the out cmdlet **1910** may be specified as out/browser; and for a window,
8 the out cmdlet **1910** may be specified as out/window. Each specific out cmdlet is
9 familiar with the capabilities of its associated destination. Locale information
10 (e.g., date & currency formats) are processed by the out cmdlet **1910**, unless a
11 convert cmdlet preceded the out cmdlet in the pipeline. In this situation, the
12 convert cmdlet processed the local information.

13 Each host is responsible for supporting certain out cmdlets, such as
14 out/console. The host also supports any destination specific host cmdlet (e.g.,
15 out/chart that directs output to a chart provided by a spreadsheet application). In
16 addition, the host is responsible for providing default handling of results. The out
17 cmdlet in this sequence may decide to implement its behavior by calling other
18 output processing cmdlets (such as format/markup/convert/transform). Thus, the
19 out cmdlet may implicitly modify sequence **1901** to any of the other sequences or
20 may add its own additional format/output cmdlets.

21 The second sequence **1902** illustrates a format cmdlet **1920** before the out
22 cmdlet **1910**. For this sequence, the format cmdlet **1920** accepts a stream of
23 pipeline objects generated and processed by other cmdlets within the pipeline. In
24 overview, the format cmdlet **1920** provides a way to select display properties and a
25 way to specify a page layout, such as shape, column widths, headers, footers, and

1 the like. The shape may include a table, a wide list, a columnar list, and the like.
2 In addition, the format cmdlet **1920** may include computations of totals or sums.
3 Exemplary processing performed by a format cmdlet **1920** is described below in
4 conjunction with FIGURE 20. Briefly, the format cmdlet emits format objects, in
5 addition to emitting pipeline objects. The format objects can be recognized
6 downstream by an out cmdlet (e.g., out cmdlet **1920** in sequence 1902) via the
7 extended type manager or other mechanism. The out cmdlet **1920** may choose to
8 either use the emitted format objects or may choose to ignore them. The out
9 cmdlet determines the page layout based on the page layout data specified in the
10 display information. In certain instances, modifications to the page layout may be
11 specified by the out cmdlet. In one exemplary process the out cmdlet may
12 determine an unspecified column width by finding a maximum length for each
13 property of a predetermined number of objects (e.g., 50) and setting the column
14 width to the maximum length. The format objects include formatting information,
15 header/footer information, and the like.

16 The third sequence **1903** illustrates a format cmdlet **1920** before the out
17 cmdlet **1910**. However, in the third sequence 1903, a markup cmdlet **1930** is
18 pipelined between the format cmdlet **1920** and the out cmdlet **1910**. The markup
19 cmdlet **1930** provides a mechanism for adding property annotation (e.g., font,
20 color) to selected parameters. Thus, the markup cmdlet **1930** appears before the
21 output cmdlet **1910**. The property annotations may be implemented using a
22 “shadow property bag”, or by adding property annotations in a custom namespace
23 in a property bag. The markup cmdlet **1930** may appear before the format cmdlet
24 **1920** as long as the markup annotations may be maintained during processing of
25 the format cmdlet 1920.

1 The fourth sequence **1904** again illustrates a format cmdlet **1920** before the
2 out cmdlet **1910**. However, in the fourth sequence **1904**, a convert cmdlet **1940** is
3 pipelined between the format cmdlet **1920** and the out cmdlet **1910**. The convert
4 cmdlet **1940** is also configured to process the format objects emitted by the format
5 cmdlet **1920**. The convert cmdlet **1940** converts the pipelined objects into a
6 specific encoding based on the format objects. The convert cmdlet **1940** is
7 associated with the specific encoding. For example, the convert cmdlet **1940** that
8 converts the pipelined objects into Active Directory Objects (ADO) may be
9 declared as “convert/ADO” on the command line. Likewise, the convert cmdlet
10 **1940** that converts the pipelined objects into comma separated values (csv) may be
11 declared as “convert/csv” on the command line. Some of the convert cmdlets
12 **1940** (e.g., convert/XML and convert/html) may be blocking commands, meaning
13 that all the pipelined objects are received before executing the conversion.
14 Typically, the out cmdlet **1920** may determine whether to use the formatting
15 information provided by the format objects. However, when a convert cmdlet
16 **1920** appears before the out cmdlet **1920**, the actual data conversion has already
17 occurred before the out cmdlet receives the objects. Therefore, in this situation,
18 the out cmdlet can not ignore the conversion.

19 The fifth sequence **1905** illustrates a format cmdlet **1920**, a markup cmdlet
20 **1930**, a convert cmdlet **1940**, and an out cmdlet **1910** in that order. Thus, this
21 illustrates that the markup cmdlet **1930** may occur before the convert cmdlet **1940**.

22 The sixth sequence **1906** illustrates a format cmdlet **1920**, a specific convert
23 cmdlet (e.g., convert/xml cmdlet **1940'**), a specific transform cmdlet (e.g.,
24 transform/xslt cmdlet **1950**), and an out cmdlet **1910**. The convert/xml cmdlet
25 **1940'** converts the pipelined objects into an extended markup language (XML)

1 document. The transform/xslt cmdlet **1950** transforms the XML document into
2 another XML document using an Extensible Style Language (XSL) style sheet. The
3 transform process is commonly referred to as extensible style language
4 transformation (XSLT), in which an XSL processor reads the XML document and
5 follows the instructions within the XSL style sheet to create the new XML
6 document.

7 The seventh sequence **1907** illustrates a format cmdlet **1920**, a markup
8 cmdlet **1930**, a specific convert cmdlet (e.g., convert/xml cmdlet **1940'**), a specific
9 transform cmdlet (e.g., transform/xslt cmdlet **1950**), and an out cmdlet **1910**.
10 Thus, the seventh sequence **1907** illustrates having the markup cmdlet **1930**
11 upstream from the convert cmdlet and transform cmdlet.

12 FIGURE 20 illustrates exemplary processing **2000** performed by a format
13 cmdlet. The formatting process begins at block 2001, after the format cmdlet has
14 been parsed and invoked by the parser and pipeline processor in a manner
15 described above. Processing continues at block 2002.

16 At block 2002, a pipeline object is received as input to the format cmdlet.
17 Processing continues at block 2004.

18 At block 2004, a query is initiated to identify a type for the pipelined
19 object. This query is performed by the extended type manager as described above
20 in conjunction with FIGURE 18. Once the extended type manager has identified
21 the type for the object, processing continues at block 2006.

22 At block 2006, the identified type is looked up in display information. An
23 exemplary format for the display information is illustrated in FIGURE 21 and will
24 be described below. Processing continues at decision block 2008.

1 At decision block 2008, a determination is made whether the identified type
2 is specified within the display information. If there is no entry within the display
3 information for the identified type, processing is complete. Otherwise, processing
4 continues at block 2010.

5 At block 2010, formatting information associated with the identified type is
6 obtained from the display information. Processing continues at block 2012.

7 At block 2012, information is emitted on the pipeline. Once the
8 information is emitted, the processing is complete.

9 Exemplary information that may be emitted is now described in further
10 detail. The information may include formatting information, header/footer
11 information, and a group end/begin signal object. The formatting information may
12 include a shape, a label, numbering/bullets, column widths, character encoding
13 type, content font properties, page length, group-by-property name, and the like.
14 Each of these may have additional specifications associated with it. For example,
15 the shape may specify whether the shape is a table, a list, or the like. Labels may
16 specify whether to use column headers, list labels, or the like. Character encoding
17 may specify ASCII, UTF-8, Unicode, and the like. Content font properties may
18 specify the font that is applied to the property values that are display. A default
19 font property (e.g., Courier New, 10 point) may be used if content font properties
20 are not specified.

21 The header/footer information may include a header/footer scope, font
22 properties, title, subtitle, date, time, page numbering, separator, and the like. For
23 example, the scope may specify a document, a page, a group, or the like.
24 Additional properties may be specified for either the header or the footer. For
25 example, for group and document footers, the additional properties may include

1 properties or columns to calculate a sum/total, object counts, label strings for totals
2 and counts, and the like.

3 The group end/begin signal objects are emitted when the format cmdlet
4 detects that a group-by property has changed. When this occurs, the format cmdlet
5 treats the stream of pipeline objects as previously sorted and does not re-sort them.
6 The group end/begin signal objects may be interspersed with the pipeline objects.
7 Multiple group-by properties may be specified for nested sorting. The format
8 cmdlet may also emit a format end object that includes final sums and totals.

9 Turning briefly to FIGURE 21, an exemplary display information **2100** is in
10 a structured format and contains information (e.g., formatting information,
11 header/footer information, group-by properties or methods) associated with each
12 object that has been defined. For example, the display information **2100** may be
13 XML-based. Each of the afore-mentioned properties may then be specified within
14 the display information. The information within the display information **2100** may
15 be populated by the owner of the object type that is being entered. The operating
16 environment provides certain APIs and cmdlets that allow the owner to update the
17 display information by creating, deleting, and modifying entries.

18 FIGURE 22 is a table listing an exemplary syntax 2201-2213 for certain
19 format cmdlets (e., format/table, format/list, and format/wide), markup cmdlets
20 (e.g., add/markup), convert cmdlets (e.g., convert/text, convert/sv, convert/csv,
21 convert/ADO, convert/XML, convert/html), transform cmdlets (e.g.,
22 transform/XSLT) and out cmdlets (e.g., out/console, out/file). FIGURE 23
23 illustrates results rendered by the out/console cmdlet using various pipeline
24 sequences of the output processing cmdlets (e.g., format cmdlets, convert cmdlets,
25 and markup cmdlets).

1 As described, the mechanism for providing data driven command line
2 output may be employed in an administrative tool environment. However, those
3 skilled in the art will appreciate that the mechanism may be employed in various
4 environments that need to display results of pipelined commands. By using the
5 output processing cmdlets in various pipeline sequences, command line users may
6 generate robust and versatile displays with minimal code. In contrast, using
7 traditional mechanisms, extensive code in the output command is needed. In
8 addition, the extensive code may not be uniform with other code in other output
9 commands. These and other limitations with the traditional mechanism for
10 displaying results are overcome by the present mechanism for providing data
11 driven command line output.

12 Although details of specific implementations and embodiments are
13 described above, such details are intended to satisfy statutory disclosure
14 obligations rather than to limit the scope of the following claims. Thus, the
15 invention as defined by the claims is not limited to the specific features described
16 above. Rather, the invention is claimed in any of its forms or modifications that
17 fall within the proper scope of the appended claims, appropriately interpreted in
18 accordance with the doctrine of equivalents.

19
20
21
22
23
24
25